

**VILNIUS UNIVERSITY**

**ŠIAULIAI ACADEMY**

BACHELOR PROGRAMME SOFTWARE ENGINEERING

**Artificial Intelligence**

**Report on**

**“Neural Network Training” task**

Student: Anna Kutova

Lecturer: Dr. Gintautas Daunys

Šiauliai, 2023

*Table of Contents*

[*Table of Contents* 2](#_Toc152889284)

[*1.* *Selecting a dataset for classification* 2](#_Toc152889285)

[*2.* *Creatating a multilayer perceptron neural network (MLP) for classification task.* 2](#_Toc152889286)

[*3.* *Preparing data for training* 3](#_Toc152889287)

[*4.* *Selecting hyperparameters* 3](#_Toc152889288)

[*5.* *Preparing a code for training* 3](#_Toc152889289)

[*6.* *Performing neural network training.* 4](#_Toc152889290)

[*7.* *Checking the performance of the neural network applying classification metrics* 4](#_Toc152889291)

[*8.* *Saving the weights of the trained network.* 4](#_Toc152889292)

[*9.* *Results* 4](#_Toc152889293)

[*Console: 5*](#_Toc152889294)

[*Analysis: 5*](#_Toc152889295)

[*10.* *Conclusion* 6](#_Toc152889296)

[*11.* *Used code* 6](#_Toc152889297)

1. **Selecting a dataset for classification**

from sklearn.neural\_network import MLPClassifier  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler  
from sklearn.metrics import classification\_report  
import pandas as pd  
  
# Loading the data  
dataset = pd.read\_csv("accent-mfcc-data-1.csv")  
  
# Splitting into features and labels  
X = dataset.drop('language', axis=1)   
y = dataset['language']

Here, we load the dataset from a CSV file using Pandas (**pd.read\_csv**). We then split the data into features (**X**) and labels (**y**). The column 'language' is dropped from features and used as the target variable.

1. **Creating a multilayer perceptron neural network (MLP) for classification task.**

# Creating MLP  
mlp = MLPClassifier(hidden\_layer\_sizes=(100, 50), max\_iter=500, activation='relu', random\_state=42)  
  
# Model training  
mlp.fit(X\_train, y\_train)  
  
# Evaluation of results  
y\_pred = mlp.predict(X\_test)  
print(classification\_report(y\_test, y\_pred))

In this step, we create an MLP classifier with specified architecture and training parameters. The model is then trained on the training data (**X\_train**, **y\_train**), and its performance is evaluated using the **classification\_report** function.

1. **Preparing data for training**

# Splitting into training and testing sets  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
  
# Normalize the data  
scaler = StandardScaler()  
X\_train = scaler.fit\_transform(X\_train)  
X\_test = scaler.transform(X\_test)

Data is split into training and testing sets using **train\_test\_split**. The features are then normalized using **StandardScaler** to bring them to a standard scale.

1. **Selecting hyperparameters**

from sklearn.model\_selection import GridSearchCV

#…

# Defining the parameters to be checked  
param\_grid = {  
 'learning\_rate\_init': [0.001, 0.01, 0.1],  
 'batch\_size': [32, 64, 128]  
}

#…

# Print the best parameters  
print("Best parameters:", grid\_search.best\_params\_)  
  
# Updating the model with the best parameters  
best\_mlp = grid\_search.best\_estimator\_

In this step, I specify a grid of hyperparameter values, crucial for optimizing the performance of my neural network, utilizing the GridSearchCV module.

The hyperparameters include the learning rate and batch size, denoted by 'learning\_rate\_init' and 'batch\_size' in the grid. After an exhaustive search, GridSearchCV identifies the optimal combination of these hyperparameters, and I print the results using the "Best parameters" statement.

Subsequently, the model is updated with these refined parameters, ensuring an improved configuration for enhanced accuracy and preventing overfitting.

1. **Preparing a code for training**

# Train the model with the best hyperparameters  
best\_mlp.fit(X\_train, y\_train)

After obtaining the best hyperparameters, the model is trained again using the updated parameters.

1. **Performing neural network training.**

# Updating the model with the best parameters  
best\_mlp = grid\_search.best\_estimator\_  
  
# Train the model with the best hyperparameters  
best\_mlp.fit(X\_train, y\_train)

1. **Checking the performance of the neural network applying classification metrics**

# Evaluate the final model results  
y\_pred\_final = best\_mlp.predict(X\_test)  
print("Final Model Results:")  
print(classification\_report(y\_test, y\_pred\_final))

The final model's performance is evaluated on the test set using the **classification\_report**.

1. **Saving the weights of the trained network.**

import numpy as np

#…

# Save the weights of the trained network  
weights\_path = "trained\_weights.npy"  
flattened\_weights = np.concatenate([layer.flatten() for layer in best\_mlp.coefs\_])  
np.save(weights\_path, flattened\_weights)  
  
print(f"Weights saved to {weights\_path}")

The weights of the trained network are flattened and saved to a file using NumPy's **np.save**. The file is named "trained\_weights.npy".

*A black background with white text

Description automatically generated*

1. **Results**

*A screenshot of a computer

Description automatically generatedConsole:*

*Analysis:*

1. **Initial Model Results:**
   * Precision: The proportion of positive identifications that were actually correct.
   * Recall: The proportion of actual positives that were correctly identified.
   * F1-score: The harmonic mean of precision and recall, providing a balanced measure.
   * Support: The number of actual occurrences of each class in the specified dataset.

**Analysis:**

* + The initial model shows good performance across different language classes, with overall accuracy of 91%.
  + Some variations in precision, recall, and F1-score exist, but the model performs reasonably well.

1. **Best Parameters:**
   * The hyperparameters chosen by GridSearchCV for the best model are a batch size of 64 and an initial learning rate of 0.01.
2. **Final Model Results:**
   * After hyperparameter tuning, the final model demonstrates improved performance.
   * Precision, recall, and F1-score are generally higher across the classes, resulting in an accuracy of 94%.
3. **Weight Saving:**
   * The trained weights of the final model are successfully saved to the file "trained\_weights.npy."

**Overall Analysis:**

* The neural network shows good ability to distinguish between language classes.
* The hyperparameter tuning process has led to improvements in model performance.
* Precision, recall, and F1-score metrics provide a comprehensive view of the model's effectiveness for each language class.
* The accuracy of 94% on the test set indicates that the model is performing well.

1. **Conclusion**

In concluding Assignment 3 on neural network training, the process involved the creation and refinement of a Multilayer Perceptron (MLP) model for language classification. The initial model demonstrated commendable accuracy at 91%, which was further improved to 94% through meticulous hyperparameter tuning. This optimization, coupled with the successful saving of trained weights, underscores a nuanced understanding of neural network intricacies. The elevated accuracy and refined metrics collectively establish the model's efficacy for accurate language classification, reflecting a proficient application of machine learning principles.

1. **Used code**

from sklearn.neural\_network import MLPClassifier  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler  
from sklearn.metrics import classification\_report  
from sklearn.model\_selection import GridSearchCV  
import pandas as pd  
import numpy as np  
  
# Loading the data  
dataset = pd.read\_csv("accent-mfcc-data-1.csv")  
  
# Splitting into features and labels  
X = dataset.drop('language', axis=1)

y = dataset['language']  
  
# Split into training and testing sets  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
  
# Normalizing the data  
scaler = StandardScaler()  
X\_train = scaler.fit\_transform(X\_train)  
X\_test = scaler.transform(X\_test)  
  
# Defining the parameters to be checked  
param\_grid = {  
 'learning\_rate\_init': [0.001, 0.01, 0.1],  
 'batch\_size': [32, 64, 128]  
}  
  
# Creating MLP  
mlp = MLPClassifier(hidden\_layer\_sizes=(100, 50), max\_iter=500, activation='relu', random\_state=42)  
  
# Model training  
mlp.fit(X\_train, y\_train)  
  
# Evaluate initial model results  
y\_pred\_initial = mlp.predict(X\_test)  
print("Initial Model Results:")  
print(classification\_report(y\_test, y\_pred\_initial))  
  
# Use GridSearchCV to explore different parameter combinations  
grid\_search = GridSearchCV(mlp, param\_grid, cv=3, scoring='accuracy')  
grid\_search.fit(X\_train, y\_train)  
  
# Print the best parameters  
print("Best parameters:", grid\_search.best\_params\_)  
  
# Updating the model with the best parameters  
best\_mlp = grid\_search.best\_estimator\_  
  
# Train the model with the best hyperparameters  
best\_mlp.fit(X\_train, y\_train)  
  
# Evaluate the final model results  
y\_pred\_final = best\_mlp.predict(X\_test)  
print("Final Model Results:")  
print(classification\_report(y\_test, y\_pred\_final))  
  
# Save the weights of the trained network  
weights\_path = "trained\_weights.npy"  
flattened\_weights = np.concatenate([layer.flatten() for layer in best\_mlp.coefs\_])  
np.save(weights\_path, flattened\_weights)  
  
print(f"Weights saved to {weights\_path}")